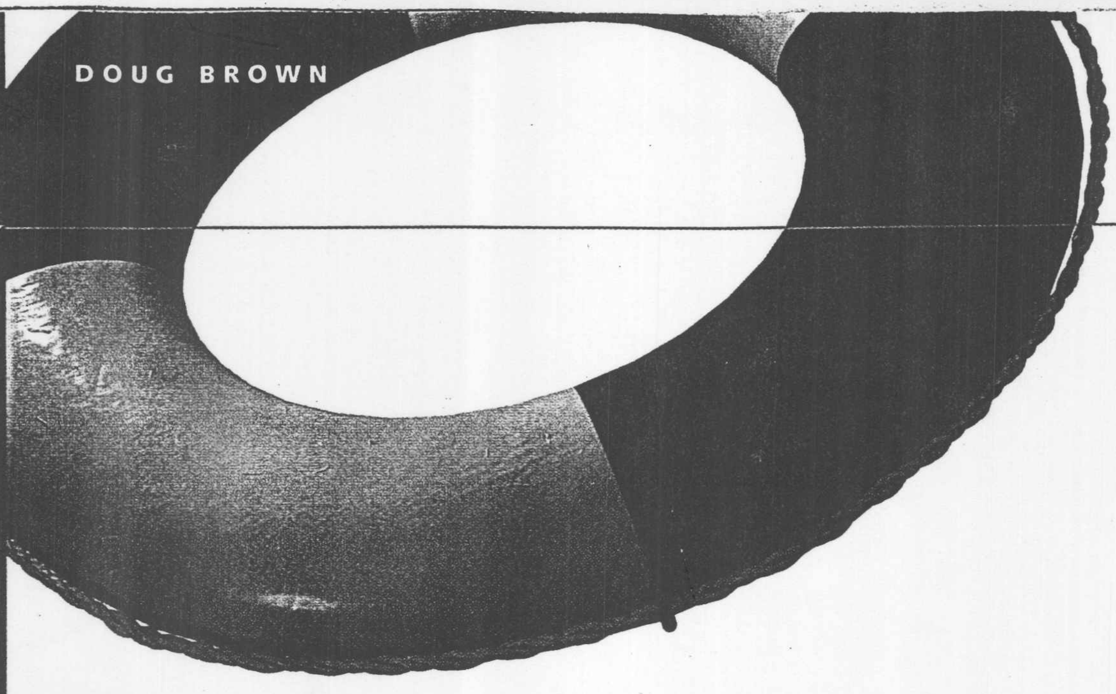
DOUG BROWN

feature



Solving the Software Safety Paradox

Embedded systems running safety-critical applications have a quandary. How can the software know that it's operating correctly? Can a malfunctioning system diagnose itself and either correct the problem or halt itself? This article examines some techniques that allow the software to monitor itself, with a bias toward those techniques that can be easily implemented in almost any embedded system.

In Joseph Heller's novel *Catch-22*, an Army Air Corp doctor faces a perplexing dilemma. One of the WWII bomber squadron's pilots is obviously insane, but he can't be grounded unless he asks to be grounded; however, if he asks to be grounded, then the rules state that this request in itself is evidence of sanity. In other words, he can't be grounded—Catch-22.

Embedded systems running safety-critical applications face a similar paradox. How can the software know that it is operating correctly, especially if it isn't? Can the system perform a sanity check on itself? This has been a major concern for manufacturers and regulators of safety-critical embedded systems since the beginning of the computer age, and it is a concern that is growing daily.

From heart defibrillators to avionics suites, from nuclear power plant controls to the anti-lock braking system in most new cars, microprocessors have become an integral part of everyday systems upon which thousands, perhaps millions, of lives depend. The safe operation of these systems cannot be taken for granted due to the potentially catastrophic consequences of a malfunction. Given the serious nature of a failure in one of these systems, is there anything we can do, beyond careful design practices and extensive testing, to ensure their correct operation? Can a malfunctioning system diagnose itself and either correct the problem or halt itself before behaving in an erroneous or potentially dangerous manner?

The answer to both questions is a qualified yes. Many simple techniques exist, which allow the software to monitor itself for correct operation. This article examines some of these techniques with a bias toward those that can be easily implemented in almost any embedded system.

CPU self-test

I have a friend who compares testing the CPU to holding a gun to your head, pulling the trigger, and then asking, "How dead am I?" While I find his analogy amusing, I disagree with it.

I admit that microprocessor failures are rare. They are so well-tested prior

How can the software know that it is operating correctly, especially if it isn't? Can the system perform a sanity check on itself?

to shipment that a CPU failure is practically unheard of. Even when they do fail, it will most likely be a catastrophic failure of the entire processor; nevertheless, it is possible for a microprocessor to fail in varying degrees, just like any other piece of hardware. In most cases, it isn't the microprocessor itself, but an external anomaly, such as a solder splash, a bent pin, or an error in the gate array interface, that causes the failure. In one instance, the CPU of a robotic welder failed, causing an extraneous jump to a `move` routine that resulted in the destruction of a small jack.[1] One can only imagine the grim result if a human had been standing in place of the jack.

So how can a software program test the CPU on which it is executing? Granted, a paradox exists in that the processor must be running well enough to execute the self-test, but by starting with very simple instructions, verifying their correct operation, and then proceeding to more complex instructions, testing the CPU is indeed possible.

The need to test individual instructions dictates that the CPU test must be written in machine assembler code instead of a higher-level language. Because branching is the main cornerstone of any program, it makes sense to start with a simple unconditional branch or jump instruction. If the branch instruction doesn't execute correctly, then the CPU is inherently unreliable. Listing 1 shows a portion of a CPU test for a 68332 microcontroller. (The complete CPU test can be found on-line at *www.embedded.com/code.htm.*) This test was written as a subroutine that can be called from C functions, so some assumptions were made that the return instruction operates correctly.

In concept, and in fact, a CPU self-test is quite simple. The `branch always` instruction should cause the microprocessor to jump around the `fail` code segment to the `cpu0` label. If the branch instruction fails, then the processor will fall into the `fail` segment and execute the failure code. You should keep the failure code fairly simple because if this segment is ever executed, the CPU is assumed to be unreliable. This segment could simply turn on an LED and then halt the processor.

Once it has been confirmed that an instruction works correctly, the subsequent tests can use the instruction. The "Branch on equal/not equal zero" test illustrates this concept. If the `beq` instruction fails to execute correctly, the processor uses the `bra` instruction to branch to the `fail` label. If the `beq` instruction executes correctly, the processor branches to the `cpu10` label and begins the next test, which can now use the `clr`, `bne`, and `beq` instructions. In this manner, each successive test builds upon the successes of the previous tests to perform evermore complex operations.

As a minimum, a CPU self-test should test each instruction that may be used by the application; verify the processor's ability to set, reset, and interrogate condition flags; verify the correct execution of math operations; and, finally, check the addressability and the read-write capabilities of each register. The register test should check for cross-talk between individual bits, bytes, and words. Use at least two bit patterns that are complementary to each other.

Considering the complexity of some of today's microprocessors, testing every possible instruction in the CPU's instruction set may not be practical. Fortunately, it's only necessary to test those instructions that may be used by the application. For example, if the CPU includes floating-point instructions but your application does not perform any floating-point operations, then it isn't necessary, although it's still a good idea, to test the floating-point operations. To determine which instructions should be tested, writing a program that parses the output of the compiler and creates a list of all the generated instructions is a fairly simple process.

CPU tests are typically run at power-on, following hardware initialization but before anything else, and during background processing. Because the test uses individual instructions, the time required to execute the test can be determined with a high degree of precision by simply adding the execution time of the assembler instructions.

In heavily regulated industries, such as medical device manufacturing, CPU self-tests also provide a substantial economic benefit. Some federal regulations require a test and inspection of each component of a device. On manufacturing lines that use thousands of microprocessors each day, the cost of testing each microprocessor before installation is prohibitive. Implementing a CPU self-test avoids the cost of implementing a separate test line, while still satisfying federal regulations.

Guarding against illegal jumps

Probably the first, and simplest, safety technique learned by many embedded programmers consists of filling unused program memory with either halt instructions or illegal instructions. This technique guards against illegal jumps outside of program space.

Consider the case where unused

RUPERT ADLEY

ROM is filled with halt instructions. If the program jumped out of bounds and executed the halt instruction, the system will stop. This is extremely helpful if it occurs in a debugging environment, since you can trace back through the stack to determine the cause of the illegal jump. In a production environment, a watchdog timer could trigger a reset of the unit.

If illegal instructions are used to fill unused program memory, a jump outside of program space would execute the illegal instruction and trigger the illegal instruction trap. If the error occurred while you're in a debug environment, the illegal instruction vector can be used to start a debugging function. If the error occurs in a production environment, the illegal instruction trap could vector to a function that attempts to "safe" the system. For heart defibrillators, the `safe` function could disable the charger and transfer relays. For robotic systems, the `safe` function could shut down all the robot's motors.

Regardless of which instruction is used, filling unused ROM provides cheap insurance against jumps outside of code space.

ROM tests

A ROM test is used to verify the integrity of a program or data stored in ROM. It is a basic safety precaution that should be performed at power-on prior to executing the main application. If the ROM test fails, then we want to avoid executing the main application because we have no idea what is programmed in the device.

In addition to performing the ROM test at power-on, running the ROM test continuously in the background makes sense in some cases. This is especially true if your device has the ability to alter its own programming (for instance, flash memory or EEPROMs).

ROM tests use a checksum or a cyclic redundancy check (CRC) to guard against corrupted memory. The technique consists of calculating the checksum or CRC over the entire ROM space, including unused memory, and storing the result in a known location. Unused ROM space should be filled with halt or illegal instructions as I discussed previously. When the test is run, boot code calculates a checksum or CRC and compares the calculated value to the stored value. If the values don't compare, then the ROM has been corrupted. The basic ROM test is simple to comprehend and implement, but some careful trade-offs need to be made when deciding on whether to use a checksum or CRC.

A checksum is simple to implement and is fast; however, it's very easy for multiple errors to cancel each other out. A CRC, on the other hand, provides a much better integrity check, but is slower. Some devices require a fast start and cannot tolerate the time required to complete a CRC check. One potential solution is to use both.

Figure 1 illustrates a program memory layout. The last eight bytes are occupied by a stored CRC value and a checksum value respectively. The checksum is computed over the entire contents of ROM, including the CRC value. At power-up, the checksum is calculated and compared to the stored value. This provides a quick integrity check that confirms the system can continue the boot process. Once the boot process is complete, a background task can run the CRC check continually. This provides a higher integrity check and also makes sure that the program ROM doesn't become corrupted during program execution.

Watchdog timers

In its simplest form, a watchdog timer[2] is a circuit or function that counts down from a preset time until it expires or is reset (tickled). The idea is for the executing software to tickle the watchdog at appropriate intervals to prevent its expiration. If the software gets caught in a loop, or is prevented from tickling the watchdog, the timer expires and causes a processor reset. In this form, a watchdog timer isn't strictly a software technique. It requires a hardware circuit to reset the processor if it isn't tickled prior to expiration; however, many real-time operating systems (RTOSes) provide a software watchdog capability that can be applied only to software tasks. On the hardware side, watchdog timers have become so common that many chips have a watchdog timer built in.

The idea is for the executing software to tickle the watchdog at appropriate intervals to prevent its expiration.

Watchdog timers are most effective in cyclic systems where the watchdog is tickled each time through the loop. Watchdogs are somewhat more difficult to implement in multitasking systems due to their non-determinism.[3] I've seen some multitasking systems that use an interrupt to tickle the watchdog. This approach defeats the whole purpose for having one in the first place. If all the tasks were blocked and unable to run, the interrupt method would continue to service the watchdog and the reset would never occur. A better solution is to use a separate monitor task that not only tickles the watchdog, but monitors the other system tasks as well. For example, a task that must run at least once per second could increment a counter. The monitor task reads the counter and "safes" the system if the task counter stops incrementing at the expected rate. If the monitor task was blocked, it would not tickle the watchdog and the system would reset.

A potential window of vulnerability exists with watchdog systems as they're implemented in most systems. Many systems only check for failure to strobe the watchdog too slow; if the system is running at a higher rate than intended, the traditional watchdog won't

LISTING 1 Portion of CPU test for a 68332 microcontroller

```
_cpu_test:

/* Branch Always Test                                            */
        bra       cpu0                   /* branch always          */
        move.l    #FALSE,d0              /* if falls through, error */
        rts                              /* exit                   */

/* Any failures from the tests below branch to this fail location.  */
fail:
        movem.l   (sp)+,d0-d7/a0-a6      /* restore registers      */
                                         /* d0 was included for testing*/
        move.l    #FALSE,d0              /* error return           */
        rts


cpu0:
        movem.l   d0-d7/a0-a6,-(sp)      /* push registers - test later*/
                                         /* d0 is included to test later */
/* Branch on equ/ne zero Test                                    */
        clr.l     d0                     /* clear a register       */
        bne       fail                   /* should not branch      */
        beq       cpu10                  /* should branch          */
        bra       fail                   /* go fail                */
cpu10:

/* Move / Compare / Branch on Conditions / Jmp Tests             */
        move.l    #LONGPOSVALUE,d0       /* set value in register  */
        cmp.l     #LONGPOSVALUE,d0       /* equal ?                */

        bne       fail                   /* should not branch      */
        bge       cpu20                  /* should branch          */
        bra       fail                   /* go fail                */
cpu20:
        bgt       fail                   /* should not branch      */
        cmp.l     #0,d0                  /* equal ?                */
        beq       fail                   /* should not branch      */
        blt       fail                   /* should not branch      */
        bgt       cpu30                  /* should branch          */
        bra       fail                   /* go fail                */
cpu30:
        bge       cpu31                  /* should branch          */
        bra       fail                   /* go fail                */
cpu31:
        jmp       cpu32                  /* should jump            */
        bra       fail                   /* go fail                */
cpu32:

/* Test movem.l and cmpm                                         */
        cmp.l     4(sp),d1               /* regs d1 - a6 must compare */
        bne       fail                   /* if no compare - fail   */
        cmp.l     8(sp),d2
```

catch it. A better technique is to use a windowed watchdog.

The windowed watchdog resets the processor if it is tickled either too fast or too slow. For example, if your task must nominally execute every 50 +/- 10% milliseconds, a windowed watchdog circuit would reset the processor if it was tickled before 45ms, or if 55ms passed without being tickled. Although windowed watchdogs are more difficult to implement, they provide a higher degree of protection than standard watchdog timers.

Redundant storage and cross checks of critical variables

All of the techniques I have discussed to this point guard against gross errors in the system, but what about more subtle errors, such as corruption of variables?

We've all heard stories of bit flips that were caused by cosmic rays or EMI. Some engineers with whom I've spoken argue that cosmic ray bit flips are rare in today's modern memories. Other engineers argue that with bit densities going up and low-voltage devices becoming more prevalent, bit flips are more likely to occur than ever. Another, and perhaps more common, cause of memory corruption is a rogue pointer, which can run wild through memory leaving a trail of corrupted variables in its wake. Regardless of the cause, the designer of safety-critical software must consider the threat that sometime, somewhere, a variable will be corrupted. If one of the corrupted variables happened to store a critical parameter, such as the energy setting for a heart defibrillator or the fluid level in a chemical mixing vat, the results could be quite unsatisfactory.

A technique commonly used in safety-critical software to guard against variable corruption is redundant storage of critical variables. Three copies of each critical variable are stored in memory. Ideally, each copy should be stored in a separate memory type

Another, and perhaps more common, cause of memory corruption is a rogue pointer, which can run wild through memory, leaving a trail of corrupted variables in its wake.

using different data formats. For example, most microprocessors have a small amount of onboard (internal) RAM in addition to the RAM designed onto the printed circuit board by the hardware designers. This provides two separate memories for storage of critical variables. One copy can be stored in internal RAM and the two other copies can be stored in the external RAM. Of the two copies in external RAM, one is stored using an inverted bit pattern.

Prior to using the variable, the three copies are compared. If any of the three variables do not agree, a two-of-three voting determines which value should be used. If none of the three values match, then a default value is used.

An extra level of protection is provided by grouping critical variables together and keeping a CRC over each group. In case of a mismatch, the CRC can be used to determine which copy is corrupted. The good copies can then be used to repair the corrupted copy.

Two drawbacks are evident when keeping redundant copies of variables. One is the additional time required to access the variable copies and perform the integrity checks. The second drawback is the shear number of variables used in embedded systems. Keeping redundant copies of all variables is impractical, so the designer must determine beforehand which variables are critical to the safe operation of the application.

LISTING 1 Portion of CPU test for a 68332 MCU

```
        bne       fail
        cmp.l     12(sp),d3
        bne       fail
        cmp.l     16(sp),d4
        bne       fail
        cmp.l     20(sp),d5
        bne       fail
        cmp.l     24(sp),d6
        bne       fail
        cmp.l     28(sp),d7
        bne       fail
        cmpa.l    32(sp),a0
        bne       fail
        cmpa.l    36(sp),a1
        bne       fail
        cmpa.l    40(sp),a2
        bne       fail
        cmpa.l    44(sp),a3
        bne       fail
        cmpa.l    48(sp),a4
        bne       fail
        cmpa.l    52(sp),a5
        bne       fail
        cmpa.l    56(sp),a6
        bne       fail
```

Stack checks

A stack check guards against stack overflow or the corruption of a task's stack. Stacks that overflow quickly are normally caught early in the development process; however, memory leaks can cause stacks to fill slowly until they eventually overflow. I once had a situation in which a stack would overflow only after several days of continuous operation. A stack-monitoring task alerted me to the problem and allowed me to correct it prior to the product shipment.

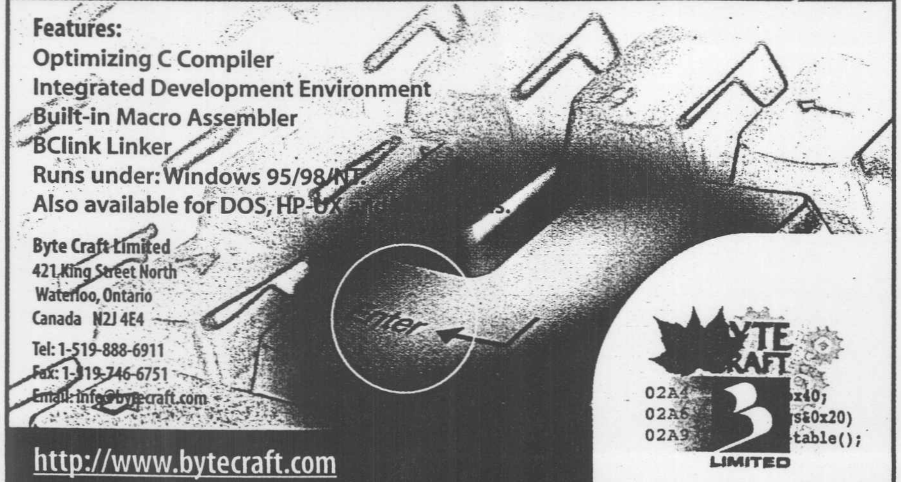
Many commercial RTOSes contain facilities and functions that support stack checking, but it can be implemented without an RTOS as well. The basic technique consists of initializing the stack with a known hex pattern. One commercial RTOS pours `COFFEE` into the stack at initialization. Although this is a cute pun, I prefer to

PROductivity Software

C COMPILERS SUPPORTING:

8051 68HC05 68HC08 COP8™ SX
MELPS740/M38000 Z8 PIC16/17Cxx

BYTE CRAFT
Limited



Features:
Optimizing C Compiler
Integrated Development Environment
Built-in Macro Assembler
BClink Linker
Runs under: Windows 95/98/NT.
Also available for DOS, HP-UX

Byte Craft Limited
421 King Street North
Waterloo, Ontario
Canada N2J 4E4
Tel: 1-519-888-6911
Fax: 1-519-746-6751
Email: info@bytecraft.com

http://www.bytecraft.com

COP8 is a trademark of National Semiconductor Corporation

CIRCLE # 24 ON READER SERVICE CARD

use a pattern that corresponds with an illegal instruction on the microprocessor. This is for the same reasons I mentioned earlier in guarding against illegal jumps.

Once the stack has been initialized with a known pattern, a simple monitor function can be written to examine the stack, determine where the pattern has been overwritten, and calculate the remaining stack space. If the stack margin shrinks below some predetermined amount, the monitor function will call an error processing routine.

Stack monitors require little overhead to implement and are an effective means of guarding against stack overflows.

Program checks

One of the primary functions of a microprocessor is computing the result of mathematical equations. The equations could be the calculations to determine the correct energy for a defibrillator shock, aircraft vectors in an air traffic control system, or control rod positions in a nuclear power plant. Whatever the purpose of the system, the correct computation of these equations is often central to its safe operation. This raises a troubling question: when doing a mathematical calculation, how do we know the result is correct?

Most software development organizations try to address this problem by extensive testing. Testing is important and should not be omitted in favor of other methods, but it does have the shortcoming of being incomplete. All possible combinations of numbers cannot be covered. Another situation occurs if your production line starts using a different, yet supposedly compatible, microprocessor. A program that used to operate correctly on the previous microprocessor fails when the upgraded part is installed. If you think this isn't a problem, remember the Pentium Floating-Point Bug? Programs that operated correctly on a 486 processor could fail when executed on a Pentium processor.

This raises a troubling question: when doing a mathematical calculation, how do we know the result is correct?

An alternative to testing is using mathematical proofs to prove that an algorithm is correct. The problem with proving correctness is the complexity involved in proving even the simplest algorithm. Additionally, a proof only proves that the algorithm is

smx® MODULAR RTOS

SYSTEM SOFTWARE OUTFITTERS

Java™

smxJVM. Try Java applets in your *smx* application at low cost! *smxJVM* includes Java Virtual Machine, Just In Time compiler, and Java libraries, including AWT. $500

Browser GUI File System RTOS TCP/IP <1/3 MB

Multitasking for x86, PowerPC, and ColdFire

smx® Full-featured, fast, small, preemptive kernel. Supports all x86, PowerPC, and ColdFire processors. x86 support for real mode (w or w/o DOS), 16-bit seg. prot. mode, and 32-bit flat prot. mode. 32-bit works with *pmEasy32*, Win95, 98 & NT. Dynamic load module support.

x86 Protected Mode

pmEasy® 16- or 32-bit protected mode entry, DPMI services, application loaders, Soft-Scope® and VisualProbe® debugger support.

DOS-Compatible File System

smxFile™ Full-featured file manager. IDE, floppy, flash, RAMdisk, PCMCIA, LS120, and SCSI drivers available. ***smxCD***. ATAPI CDROM file system.

User Interface

PEG™ Portable Embedded GUI. **PEGmt** adds multitasking. Support for **SciTech MGL**™, **MetaWINDOW**™ and **Zinc**® graphics. ***smxWindows***™ Text windowing library.

Networking

smxNet™ TCP/IP. No copy. ICMP, ARP, BOOTP, Ethernet, SLIP, and PPP drivers. FTP, NFS, Telnet, SNMP, DHCP, Web server, HTML v3.2 Graphical Browser, SMTP/POP3 email.

Debugging

smxProbe™ Provides tracing and task debugging. ***smxAware***™ & **DEBUG/*smx***™ add *smx*-awareness to debuggers. ***WinBase*** permits IDE development & debugging at low cost.

C++ Classes

smx++™ Class library built upon *smx*. Provides complete support for embedded C++.

DOS & Windows Emulation

X-DOS® Full v5 DOS. ***unDOS***™ DOS, BIOS, and Windows emulator. **EXE Bootloader** boots from BIOS.

Tool & Board Support from A-Z

AMD	MetaGraphics	SciTech
Beacon	MetaWare	SDS
Borland	MetroWerks	Transvirtual
Diab	Microsoft	VersaLogic
EBS	National	Watcom
Intel	Paradigm	Zinc

NO ROYALTIES • 6 MO FREE SUPPORT • 30-DAY FREE TRIAL

MICRO DIGITAL 1-800-366-2491 www.smxinfo.com/rjes.htm

CIRCLE # 25 ON READER SERVICE CARD

To this day, I can still hear Mrs. Stewart admonishing my second grade class to "Check your work."

correct. It cannot catch processor errors.

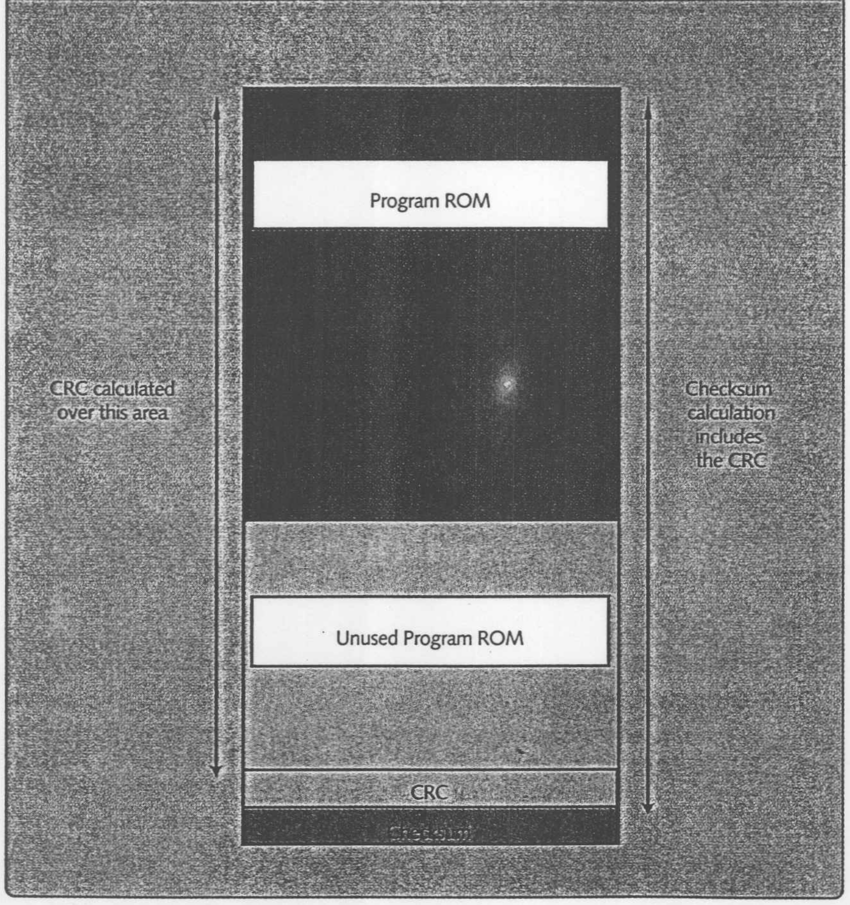
A third, and probably the best alternative, is to use program checks[4] to verify calculation results. Program checks address the basic problem of verifying that a calculation is correct before the result is utilized. The technique is one that you were taught in elementary school.

To this day, I can still hear Mrs. Stewart admonishing my second grade class to "Check your work." Check subtraction by adding the result to one of the numbers in the original problem to get the other number. (for example, If 10 – 3 = 7 then 7 + 3 had better equal 10.) Check division by plugging your answer back into a multiply, and so on. In their simplest form, program checks are the same thing you were taught in grade school.

When checking calculations, be sure to allow tolerances for checking your answers. Digital arithmetic isn't exact! The designer must decide on an error tolerance and only perform the error processing if the result falls outside of this tolerance.

Program checks have two drawbacks. First, program checks roughly double your computation time because each calculation is essentially performed twice: once for the original calculation, and again, in reverse, to check the result. In a system where processor throughput is an issue, checking all of the calculations may not be practical. Second, if the check signals a mismatch, we're not sure where the error has occurred, whether in the original calculation or in the check calculation. We only know that there is a problem and further analysis is necessary to determine the cause. Despite these drawbacks, program checks offer a powerful means for programs to check themselves for correct operation.

FIGURE 1 Program memory layout


Program ROM
Unused Program ROM
CRC
Checksum
CRC calculated over this area
Checksum calculation includes the CRC

Simple, yet effective

These safety self-tests I've described are only a few of the techniques that you can employ to have software check itself for proper execution—many others exist as well. The techniques here are some of the simplest, yet most effective. They can be easily implemented in most embedded systems. Although these safety self-tests can detect a bad microprocessor, bad ROM, or wild software, they certainly are not a guarantee of correct operation. They can't diagnose design flaws, nor can they compensate for requirements errors. Nevertheless, they are effective tools in designing safe systems and are a welcome addition to a programmer's toolbox. **esp**

Doug Brown is a senior project engineer at Physio-Control. Contact him at ddbrown@physio-control.com

References

1. Neumann, Peter G. *Computer Related Risks*. Reading, MA: Addison Wesley, 1995, Chapter 2.8.1, p. 66.
2. Santic, John S., "Watchdog Timer Techniques," *Embedded Systems Programming*, April 1995, p. 58.
3. Lowell, Augustus P., "The Care and Feeding of Watchdogs," *Embedded Systems Programming*, April 1992, p. 38.
4. Blum, Manuel and Sampath Kannan, "Designing Programs that Check Their Work," *Journal of the Association for Computing Machinery*, January 1995, p. 269.